Programmeerimise põhikursus

ITI0010

Loengu sisu

Sissejuhatus praktikumiülesandesse nr 2

- Materjalid kursuse saidil

Veatöötlus:

- Miks veatöötlus
- Catch ja throw ja sõbrad

Sisend/väljund

- Failid
- Võrk: lihtsamad viisid võrgust lugeda

Robustness and error handling

A robust program is one that can survive unusual or "exceptional" circumstances without crashing.

For example, a program will crash if it tries to use an array element `A[i]`, when `i` is not within the declared range of indices for the array `A`.

A robust program must anticipate the possibility of a bad index and guard against it.

Increasing robustness: preconditions

For example:

```
if (i < 0 || i >= A.length) {
        ...  // Do something to handle the
             // out-of-range index, i
 }
 else {
     System.out.println(A[i]);
 }
```

We would say that the statement "`System.out.println(A[i]);`" has the precondition that `i >= 0 && i < A.length`.

A precondition is a condition that must be true at a certain point in the execution of a program, if that program is to continue without error and give a correct result.

Preconditions

One approach to writing robust programs is to rigorously apply the rule, "At each point in the program, identify any preconditions and make sure that they are true" -- either by using `if` statements to check whether they are true, or by verifying that the required preconditions are consequences of what the program has already done.

```
int index;

while (true) {
   TextIO.put("Which array element do you want to see? ");
   index = TextIO.getlnInt();
   if (index >= 0 && index < A.length)
      break;
    TextIO.put("Your answer must be >= 0 and < " + A.length);
}

// At this point, we can be absolutely sure that the value
// of index is in the legal range of indices for the array A.

TextIO.putln("A[" + index + "] is " + A[i]);
```

Problems with the approach

It is difficult and sometimes impossible to anticipate all the possible things that might go wrong.

Trying to anticipate all the possible problems can turn what would otherwise be a straightforward program into a messy tangle of `if` statements.

Exception handling

Java and C++ provide a neater, more structured alternative method for dealing with possible errors that can occur while a program is running.

The method is referred to as exception-handling. The word "exception" is meant to be more general than "error." It includes any circumstance that arises as the program is executed which is meant to be treated as an exception to the normal flow of control of the program.

An exception might be an error, or it might just be a special case that you would rather not have clutter up your elegant algorithm.

Throw and catch

When an exception occurs during the execution of a program, we say that the exception is thrown.

When this happens, the normal flow of the program is thrown off-track, and the program is in danger of crashing.

However, the crash can be avoided if the exception is caught and handled in some way.

An exception can be thrown in one part of a program and caught in a completely different part. An exception that is not caught will generally cause the program to crash.

More exactly, the thread that throws the exception will crash. In a multithreaded program, it is possible for other threads to continue even after one crashes.

Exception object

When an exception occurs, it is actually an object that is thrown. This object can carry information (in its instance variables) from the point where the exception occurred to the point where it is caught and handled.

This information typically includes an error message describing what happened to cause the exception, but it could also include other data.

The object thrown by an exception must be an instance of the class `Throwable` or of one of its subclasses.

In general, each different type of exception is represented by its own subclass of `Throwable`. `Throwable` has two direct subclasses,

`Error`

`Exception`.

These two subclasses in turn have many other predefined subclasses. In addition, a programmer can create new exception classes to represent new types of exceptions.

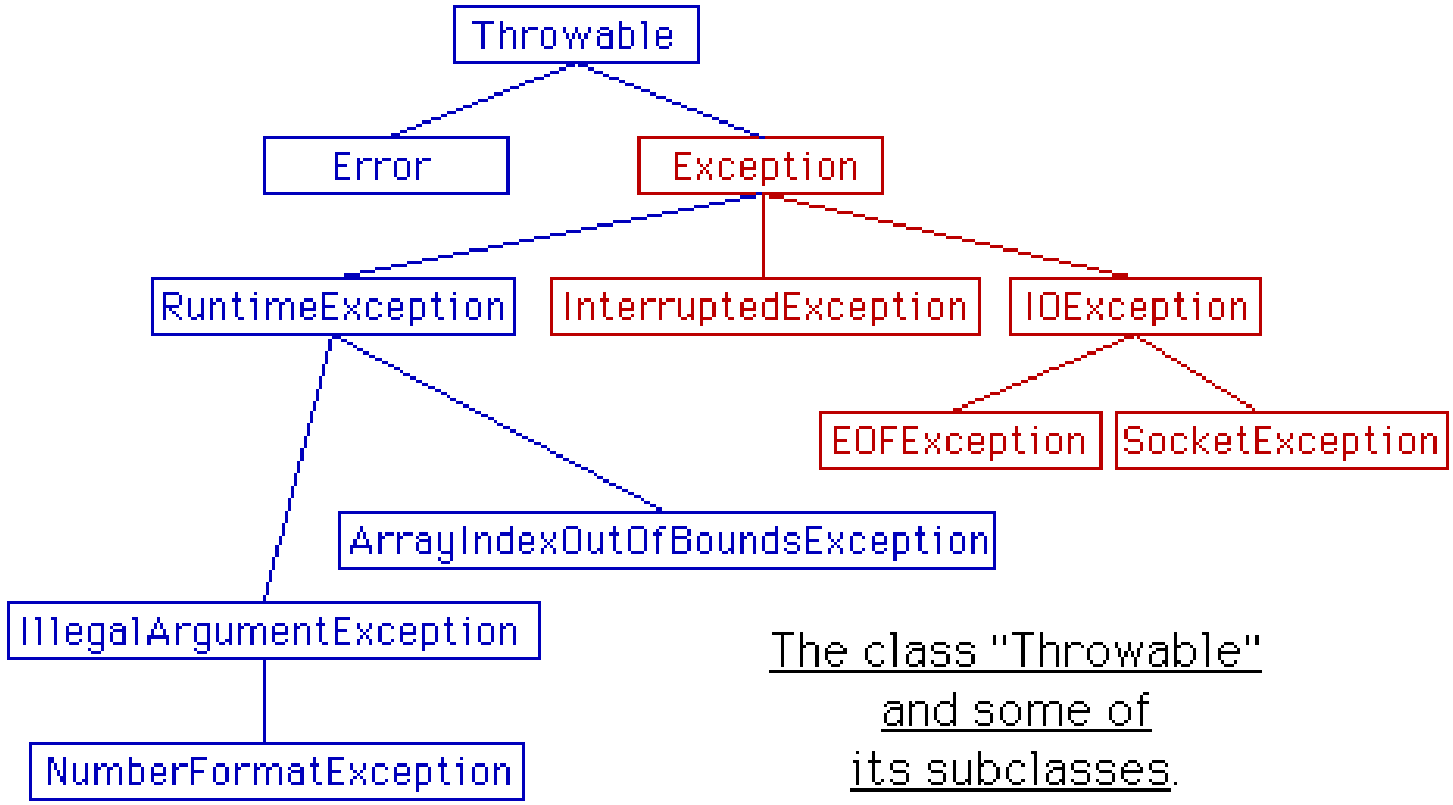
Error and exception - pragmatical difference

Most of the subclasses of the class **`Error`** represent serious errors within the Java virtual machine that should ordinarily cause program termination because there is no reasonable way to handle them.

Subclasses of **`Exception`** represent exceptions that are meant to be caught. In many cases, these are exceptions that might naturally be called "errors," but they are errors in the program or in input data that a programmer can anticipate and possibly respond to in some reasonable way.

Exception class hierarchy

The following diagram is a class hierarchy showing the class `Throwable` and just a few of its subclasses. Classes that require mandatory exception-handling are shown in red.



Handle exceptions!

To handle exceptions in a Java program, you need a `try` statement. The idea is that you tell the computer to "try" to execute some commands. If it succeeds, all well and good. But if an exception is thrown during the execution of those commands, you can catch the exception and handle it. For example,

```
try {
      double determinant = M[0][0]*M[1][1] -
                           M[0][1]*M[1][0];
      System.out.println("The determinant of M is " +
                         determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
     System.out.println("M is the wrong size to have a
                        determinant.");
}
```

Multiple catches

You might notice that there are is another possible source of error in this `try` statement. If the value of the variable `M` is `null`, then a `NullPointerException` will be thrown when the attempt is made to reference the array `M`.

```
try {
   double determinant = M[0][0]*M[1][1] -
                        M[0][1]*M[1][0];
   System.out.println("The determinant of M is " +
                      determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
  System.out.println("M is the wrong size to have a
                      determinant.");
}
catch ( NullPointerException e ) {
  System.out.println("Programming error!  M doesn't
        exist. " + e.getMessage());
}
```

Exception object is passed to a catch

This example shows what that little "e" is doing in the catch clauses. The e is actually a variable name. Recall that when an exception occurs, it is actually an object that is thrown. Before executing a catch clause, the computer sets this variable to refer to the exception object that is being caught.

This object contains information about the exception. In particular, every exception object includes an error message, with can be retrieved using the object's `getMessage()` method, as is done in the above example.

Syntax of a try

The syntax can be described as

```
try {
    statements
}
optional-catch-clauses
optional-finally-clause
```

The syntax for a `catch` clause is

```
catch ( exception-class-name variable-name ) {
      statements
}
```

and the syntax for a `finally` clause is

```
finally {
      statements
}
```

Finally

The semantics of the finally clause is that the block of statements in the finally clause is guaranteed to be executed as the last step in the execution of the try statement, whether or not any exception occurs and whether or not any exception that does occur is caught and handled.

The finally clause is meant for doing essential cleanup that under no circumstances should be omitted.

Deliberate exceptions

There are times when it makes sense for a program to deliberately throw an exception.

This is the case when the program discovers some sort of exceptional or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception.

To throw an exception, use a `throw` statement. The syntax of the `throw` statement is

```
throw exception-object;
```

The exception-object must be an object belonging to one of the subclasses of `Throwable`. Usually, it will in fact belong to one of the subclasses of `Exception`. In most cases, it will be a newly constructed object created with the `new` operator. For example:

```
throw new ArithmeticException("Division by zero");
```

Exception throwing

A subroutine that can throw an exception can announce this fact by adding the phrase "`throws` **exception-class-name**" to the specification of the routine. For example:

```
static double root(double A, double B, double C)
                  throws IllegalArgumentException {
      // returns the larger of the two roots of
      // the quadratic equation A*x*x + B*x + C = 0
     if (A == 0) {
         if (B == 0)
            throw new IllegalArgumentException("A and B
                  can't both be zero.");
            return  -C / (2*B);
      } else {
         double disc = B*B - 4*A*C;
         if (disc < 0)
             throw new IllegalArgumentException("Discriminant

                   less than zero.");
         return  (-B + Math.sqrt(disc)) / (2*A);
     }
}
```

Mandatory/non-mandatory handling

In this example, declaring that `root()` can throw an `IllegalArgumentException` is just a courtesy to potential readers of this routine. This is because handling of `IllegalArgumentExceptions` is not mandatory. A routine can throw an `IllegalArgumentException` without announcing the possibility. And the user of such a routine is free either to catch or ignore such an exception.

For those exception classes that require mandatory handling, the situation is different. If a routine can throw such an exception, that fact must be announced in a `throws` clause in the routine definition. Failing to do so is a syntax error that will be reported by the compiler.

I/O

The interaction of a program with the rest of the world is referred to as input/output or I/O.

A computer can be connected to many different types of input and output devices.

One of the major achievements in the history of programming has been to come up with good abstractions for representing I/O devices. In Java, the I/O abstractions are called streams.

Machine- and human-readable streams

There are two broad categories of data: machine-formatted data and human-readable data.

Machine-formatted data is represented in the same way that data is represented inside the computer, that is, as strings of zeros and ones.

Human-readable data is in the form of characters. When you read a number such as 3.141592654, you are reading a sequence of characters and interpreting them as a number.

To deal with the two broad categories of data representation, Java has two broad categories of streams:

byte streams for machine-formatted data

character streams for human-readable data.

There are many predefined classes for representing streams of each type.

Byte streams for machines

Every object that outputs data to a byte stream belongs to one of the subclasses of the abstract class `OutputStream`.

Objects that read data from a byte-stream belong to subclasses of `InputStream`. If you write numbers to an `OutputStream`, you won't be able to read the resulting data yourself. But the data can be read back into the computer with an `InputStream`.

The writing and reading of the data will be very efficient, since there is no translation involved: the bits that are used to represent the data inside the computer are simply copied to and from the streams.

Character streams for humans

For reading and writing human-readable character data, the main classes are **`Reader`** and **`Writer`**.

All character stream classes are subclasses of one of these.

If a number is to be written to a `Writer` stream, the computer must translate it into a human-readable sequence of characters that represents that number. Reading a number from a `Reader` stream into a numeric variable also involves a translation, from a character sequence into the appropriate bit string.

Even if the data you are working with consists of characters in the first place, such as words from a text editor, there might still be some translation.

Characters are stored in the computer as 16-bit Unicode values. For people who use Western alphabets, character data is generally stored in files in ASCII code, which uses only 8 bits per characters. The `Reader` and `Writer` classes take care of this translation.

Stream classes

All the stream classes are defined in the package `java.io`, along with several supporting classes. You must `import` the classes from this package if you want to use them in your program.

Streams are not used in Java's graphical user interface, which has its own form of I/O. But they are necessary for working with files (using the classes `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter`) and for doing communication over a network. They can be also used for communication between two concurrently running threads.

The beauty of the stream abstraction is that it is as easy to write data to a file or to send data over a network as it is to print information on the screen.

Wrapping streams

Java's I/O package is that it lets you add capabilities to a stream by "wrapping" it in another stream object that provides those capabilities.

The wrapper object is also a stream, so you can read from or write to it -- but you can do so using fancier operations than those available for basic streams.

Objects that can be used as wrappers in this way belong to subclasses of `FilterInputStream` and `FilterOutputStream` for byte streams, and to subclasses of `FilterReader` and `FilterWriter` for character streams.

By writing new subclasses of these classes, you can make your own I/O filters to provide any style of I/O that you want.

Examples

For example, if `charSink` is of type `Writer` then you could say

```
PrintWriter printableCharSink = new PrintWriter(charSink);
```

When you output data to `printableCharSink`, using `PrintWriter's` advanced data output methods, that data will go to exactly the same place as data written directly to `charSink`.

The output methods of the `PrintWriter` class include:

```
public void print(String s)    // methods for outputting
public void print(char c)      //    standard data types
public void print(int i)       //    to the stream, in
public void print(long l)      //    human-readable form.
       ...
public void println()    // output a carriage return
public void println(String s)
public void println(int i)
       ...
```

Files

For permanent storage, computers use files, which are collections of data stored on the computer's hard disk, on a floppy disk, on a CD-ROM, or on some other type of storage device.

Files are organized into directories (sometimes called "folders"). A directory can hold other directories, as well as files. Both directories and files have names that are used to identify them.

Programs can read data from existing files. They can create new files and can write data to files.

File I/O via streams

In Java, input and output is done using streams.

Human-readable character data is read from a file using an object belonging to the class `FileReader`, which is a subclass of `Reader`.

Similarly, data is written to a file in human-readable format through an object of type `FileWriter`, a subclass of `Writer`.

For files that store data in machine-readable format, the appropriate I/O classes are `FileInputStream` and `FileOutputStream`.

NB! Restrictions!

Applets which are downloaded over a network connection are generally not allowed to access files. This is a security consideration.

Standalone programs written in Java, however, have the same access to your files as any other program. When you write a standalone Java application, you can use all the file operations.

Open input stream for the file

The `FileReader` class has a constructor which takes the name of a file as a parameter and creates an input stream that can be used for reading from that file.

For example, suppose you have a file named "`data.txt`", and you want your program to read data from that file. You could do the following to create an input stream for the file:

```
FileReader data;
try {
   data = new FileReader("data.txt"); // create
                                      // the stream
}
catch (FileNotFoundException e) {
   ... // do something to handle the error --
       // maybe, end the program
}
```

Option 1: read from the stream directly

Once you have successfully created a `FileReader`, you can start reading data from it.

However, there are only two ways to read from `FileReader`:

Read a single character at a time using

read()

Read N characters into a character array using

read(char[] cbuf, int offset, int length)

```
...
char c = data.read()
...
```

This may be inconvenient, sometimes

Option 2: read from the stream using TextReader

Since `FileReaders` have only the primitive input methods inherited from the basic `Reader` class, you may want to **wrap your `FileReader` in a `TextReader` object or in some other wrapper class**.

```
TextReader data;
try {
   data = new TextReader(new FileReader("data.txt"));
} catch (FileNotFoundException e) {
    ... // handle the exception
}
```

Once you have a `TextReader` named `data`, you can read from it using such methods as **`data.getInt()`** and **`data.getWord()`**, exactly as you would from any other `TextReader`.

NB! `TextReader` is a class written by Eck

Option 3: read the stream using BufferedReader

You may want to wrap your `FileReader` in a BufferedReader, which is present in Java library (ie no need to use a class written by Eck)

```
BufferedReader data;
try {
    data = new BufferedReader(new FileReader("data.txt"));
} catch (FileNotFoundException e) {
    ... // handle the exception
}
```

Once you have a `BufferedReader` named `data`, you can read from it using also a method **`data.readLine()`** which gives you one line (a string) at a time

Raw byte- and byte-array reading method **`data.read()`** is also available

Summary: reading from the stream

Basic stream classes like **FileReader** give you only a few basic options to read data, typically using

read(), to read data byte-by-byte or N bytes into an array.

A bit fancier “wrapper” classes like **BufferedReader** add new methods to read, like readLine(), which allow to read strings (one line is one string)

You may use even fancier wrapper classes written by other people (like **TextReader**, which allows to read integers, floats etc) or write your own classes for reading.

To summarise, **there are quite many options for reading**: you have to pick a suitable wrapper class around the basic input class.

Output is similar: basic output plus fancy wrappers

For example, suppose you want to write data to a file named "`result.dat`". Since the constructor for `FileWriter` can throw an exception of type `IOException`, you should use a `try` statement:

```
PrintWriter result;


try {
   result = new PrintWriter(new Filewriter("result.dat"));
}
catch (IOException e) {
    ... // handle the exception
}
```

If no file named `result.dat` exists, a new file will be created. If the file already exists, then the current contents of the file will be erased and replaced with the data that your program writes to the file. An `IOException` might occur if, for example, you are trying to create a file on a disk that is "write-protected," meaning that it cannot be modified.

Closing a file

After you are finished using a file (for reading ow writing), it's a good idea to close the file, to tell the operating system that you are finished using it.

```
data.close();
```

If you forget to do this, the file will probably be closed automatically when the program terminates or when the file stream object is garbage collected, but it's best to close a file as soon as you are done with it.

You can close a file by calling the `close()` method of the associated file stream. Once a file has been closed, it is no longer possible to read data from it or write data to it, unless you open it again as a new stream.

Example pg 1

```
import java.io.*;
public class ReverseFile {
    public static void main(String[] args) {
        TextReader data;
        PrintWriter result;
        double[] number = new double[1000];
        int numberCt;
        try {  // create the input stream
           data = new TextReader(new FileReader("data.dat"));
        }
        catch (FileNotFoundException e) {
              System.out.println("Can't find file
  data.dat!");
              return;  // end the program
        }
```

Example pg 2

```
try {  // create the output stream
   result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
   System.out.println("Can't open file result.dat!");
   System.out.println(e.toString());
   data.close();  // close the input file
  return;  // end the program
}

try {
   // read the data from the input file,
   numberCt = 0;
   while (!data.eof()) {  // read to end-of-file
         number[numberCt] = data.getlnDouble();
         numberCt++;
}
```

Example pg 3

```
    // then output the numbers in reverse order
    for (int i = numberCt-1; i >= 0; i--)
        result.println(number[i]);

    System.out.println("Done!");
    } catch (TextReader.Error e) {
        System.out.println("Input Error: " +
  e.getMessage());
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Too many numbers in data
  file.");
        System.out.println("Processing has been aborted.");
    } finally {
        data.close();
        result.close();
    }
  }  // end of main()
} // end of class
```

File names and dialogs

To fully specify a file, you have to give both the name of the file and the name of the directory where that file is located.

A simple file name like "data.dat" or "result.dat" is taken to refer to a file in a directory that is called the current directory (or "default directory" or "working directory").

The current directory is not a permanent thing. It can be changed by the user or by a program. Files not in the current directory must be referred to by a path name, which includes both the name of the file and information about the directory where it can be found.

There are two types of path names, absolute path names and relative path names. An absolute path name uniquely identifies one file among all the files available to the computer. It contains full information about which directory the file is in and what its name is. A relative path name tells the computer how to locate the file, starting from the current directory.

Examples of file names

- `data.dat` -- on any computer, this would be a file named data.dat in the current directory.
- `/home/eck/java/examples/data.dat` -- This is an absolute path name in the UNIX operating system. It refers to a file named data.dat in a directory named examples, which is in turn in a directory named java,....
- `C:\eck\java\examples\data.dat` -- An absolute path name on a DOS or Windows computer.
- `Hard Drive:java:examples:data.dat` -- Assuming that "Hard Drive" is the name of a disk drive, this would be an absolute path name on a Macintosh computer.
- `examples/data.dat` -- a relative path name under UNIX. "Example" is the name of a directory that is contained within the current directory, and data.data is a file in that directory. The corresponding relative path names for Windows and Macintosh would be `examples\data.dat` and `examples:data.dat`.

File dialog box

For example, if you want the user to select a file for output, and if the main window for your application is `mainWin`, you could say:

```
FileDialog fd = new FileDialog(mainWin,
                               "Select output file",
                               FileDialog.SAVE);
fd.show();
String fileName = fd.getFile();
String directory = fd.getDirectory();
if (fileName != null) {  // (otherwise, the user
                         // canceled the request)
       ...  // open the file, save the data, then
            // close the file

}
```

Once you have the file name and directory information, you will have to combine them into a usable file specification. The best way to do this is to create an object of type File.

Http stream: one way to read from internet

```
import java.net.*;
import java.io.*;

public class URLConnectionReader {
     public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
                                    new InputStreamReader(
                                        yc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
     }
}
```